NOTES DE COURS (01)

Mercredi 16 mars 2005

SÉMANTIQUE FORMELLE

Contenu de la partie (d'après [Meyer90])

- Introduction
- Syntaxe abstraite
- Lambda-calcul
- Sémantique dénotationnelle
- Récursivité
- Sémantique axiomatique

[Meyer90] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice Hall, 1990.
Bertrand Meyer. *Introduction à la théorie des langages de programmation*, InterEditions, 1992.

INTRODUCTION

Contenu du chapitre

- Rôle des descriptions formelles
- Différence entre syntaxe et sémantique
- Différences entre notations mathématiques et programmation
- Métalangage

DESCRIPTIONS FORMELLES

Utilité des descriptions formelles

- faciliter la compréhension des langages
- faciliter la conception et la spécification d'un langage
- faciliter la normalisation des langages
- faciliter l'écriture des compilateurs
- faciliter la spécification du logiciel
- faciliter la vérification et la validation des programmes

Limites des descriptions formelles

- lourdeur et complexité de la problématique
- inadéquation pour des gros langages ... ?

SYNTAXE ET SÉMANTIQUE

Liens entre syntaxe et sémantique

- La syntaxe décrit la structure des programmes
- La sémantique caractérise le comportement du programme; on distingue
  - la sémantique statique
  - la sémantique dynamique
- La description de la sémantique s'appuie sur la syntaxe

MATHÉMATIQUES ET PROGRAMMATION

Éléments impératifs

- La plupart des langages de programmation sont *impératifs* et peuvent provoquer des *effets de bords*; ils sont de ce fait plus difficiles à décrire
  - en mathématique:

    $x = e$

    affirme l'égalité entre deux objets
  - en programmation

    $x := e$

    provoque un changement d'état, mais ne garantit pas de propriété
- Les langages de programmation purement fonctionnels n'ont pas ce problème; mais leur utilité pratique est limitée.

Transparence référentielle

- La propriété suivante est appelée *transparence référentielle*

  si `a=b` et `p(a)` sont vrais alors `p(b)` est vrai
- Les formalismes mathématiques satisfont la transparence référentielle

  `x=2, x+y>5 => y>3`
- Les langages de programmation avec effet de bord ne satisfont pas la transparence référentielle

```
var y: Integer;
function f(var x: integer): integer
begin y:=y+1; f:=y+x end;
```

`2*f(1)<>f(1)+f(1)`

`y:=0; z:=0; f(y)=2` mais `y:=0; z:=0; f(z)=1`

Transparence référentielle

- L'objectif de la sémantique formelle est de décrire des constructions référentiellement opaques au moyen de constructions référentiellement transparentes.

MÉTALANGAGE

Définition

- Le formalisme utilisé pour décrire les langages est appelé *métalangage*.

Problèmes

- Il faut éviter des description circulaires.
- Il ne faut confondre les notations du métalangage avec celles du langage décrit.

SYNTAXE ABSTRAITE

Contenu du chapitre

- Différence entre syntaxe concrète et syntaxe abstraite
- Grammaires abstraites
- Expressions syntaxiques abstraites
- Spécification de la syntaxe concrète
- Principes pour la formalisation de la sémantique

SYNTAXE CONCRÈTE ET SYNTAXE ABSTRAITE

Syntaxe concrète

- elle décrit l'apparence externe des constructions d'un langage
- exemple (grammaire BNF):

```
<conditionnelle> ::= if <expression> then <instruction>
                     else <instruction>
```

- elle est peu adaptée pour la description formelle de la sémantique

Syntaxe abstraite

- elle décrit la structure interne des constructions d'un langage
- exemple (en langage Métanot):

```
Conditional ≙ thenBranch : Instruction ;
              elseBranch : Instruction ;
              test : Boolean_Expression
```

NOTATION

Définition vs égalité

- L'opérateur $\stackrel{\Delta}{=}$ est utilisé pour distinguer les définitions de l'opérateur relationnel d'égalité:

  `x` $\stackrel{\Delta}{=}$ `e`

  signifie que le nom `x` "est défini comme" étant l'expression `e`
- on suppose que l'expression `e` n'utilise que des termes dont la définition est indépendante de `x` (principe de la non créativité des définitions):

  `x` $\stackrel{\Delta}{=}$ `x+1` n'a pas de sens

GRAMMAIRES ABSTRAITES

Terminologie (d'après [Meyer90])

- une *grammaire abstraite* est un formalisme qui décrit la syntaxe abstraite d'un langage de programmation;
- une *construction* définit de façon générique la structure d'un ensemble d'objets;
- les instances d'une construction sont appelées *spécimens*;
- la construction associée à un spécimen est appelée *type syntaxique*;
- la *construction racine* est celle qui représente les objets de niveau le plus élevé
- l'ensemble des spécimens de toutes les constructions forme le *langage* ; (selon l'approche usuelle, il serait formé uniquement des spécimens de la construction racine!)

Notation MÉTANOT

- Une grammaire MÉTANOT contient
  - un ensemble fini de constructions représentées par des identificateurs (par convention, commençant par une majuscule);
  - un ensemble fini de productions de la forme

    `Construction` $\stackrel{\Delta}{=}$ `partie_droite`

    chaque production est associée à une construction; inversement il existe au plus une production par construction;
  - les constructions qui ont une production associée sont dites non terminales; les autres sont dites terminales;
- MÉTANOT comprend trois catégories de productions
  - l'*agrégat* définit une construction dont les spécimens sont formés d'un nombre fixe de composants

    ```
    Conditional ≙ thenBranch, elseBranch: Instruction;
                  test: Expression
    ```

  - l'ordre des composants n'est pas significatif
  - un agrégat peut être formé d'un seul composant
- le *choix* définit une construction dont les spécimens ont un type syntaxique choisi parmi plusieurs autres constructions

  ```
  Instruction ≙ Skip | Assignment |
                Compound | Conditional | Loop
  ```

- la *liste* définit une construction dont les spécimens sont formés de zéro, un ou plusieurs composants d'un même type syntaxique

  ```
  Compound ≙ Instruction*
  ```

  - une liste contenant au moins un composant est noté

    ```
    Number ≙ Digit+
    ```

- MÉTANOT comprend quatre constructions terminales
  - **N** : l'ensemble des nombre naturels
  - **Z** : l'ensemble des nombres entiers
  - **B** : l'ensemble formé des valeurs logiques, vrai et faux
  - **S** : l'ensemble des chaînes de caractères

Grammaire de GRAAL (Great Relief After Ada Lessons)

```
Program          ≙ decpart: Declaration_list ;
                   body: Instruction
Declaration_list ≙ Declaration*
Declaration      ≙ v: Variable; t: Type
Type             ≙ Boolean_type | Integer_type
Instruction      ≙ Skip | Assignment | Compound |
                   Conditional | Loop
Assignment       ≙ target: Variable; source: Expression
Compound         ≙ Instruction*
Conditional      ≙ thenbranch, elsebranch: Instruction
                   test: Expression
Loop             ≙ body: Instruction; test: Expression
Expression       ≙ Constant | Variable | Binary
Constant         ≙ Integer_constant | Boolean_constant
Binary           ≙ term1, term2: Expression; op: Operator
Operator         ≙ Boolean_op | Relational_op | Arithmetic_op
Boolean_op       ≙ And | Or | Nand | Nor | Xor
Relational_op    ≙ Lt | Le | Eq | Ne | Ge | Gt
Arithmetic_op    ≙ Plus | Minus | Times | Div
Boolean_constant ≙ value: B
Integer_constant ≙ value: Z
Variable         ≙ id: S
```

Exercice

Décrire la syntaxe abstraite de MÉTANOT dans le langage MÉTANOT lui-même.

EXPRESSIONS SYNTAXIQUES ABSTRAITES

Introduction

- Les expressions syntaxiques abstraites servent à décrire les spécimens d'une grammaire abstraite
- La forme des expressions syntaxiques abstraites est formalisée pour chaque catégorie de productions

Expressions relatives aux agrégats

Considérons une production de la catégorie des agrégats, par exemple

```
Program ≜ decpart: Declaration_list ; body: Instruction
```

- pour décrire un spécimen à partir de ces composants, on écrit:

  ```
  p ≜ Program (decpart: dl; body: instr)
  ```

  où `dl` et `instr` représentent des spécimens de type `Declaration_list` et de type `Instruction`;
- pour décrire un composant d'un spécimen de la catégorie agrégat, on écrit

  ```
  p.decpart
  p.body
  ```

  où `p` représente un spécimen de type `Program`.

Expressions relatives aux choix

Considérons une production de la catégorie des choix, par exemple

```
Instruction ≜ Skip | Assignment | Compound | Conditional |
              Loop
```

- pour décrire un spécimen à partir d'une alternative, on écrit

  ```
  i ≜ Instruction (ass)
  ```

  où `ass` représente par exemple un spécimen de type `Assignment`;
- pour décrire un spécimen d'un type syntaxique résultant d'un choix, on écrit

```
case i of
   Skip => ... |
   Assignment => ass |
   Compound => ... |
   ...
end
```

NOTATIONS

Manipulation de listes

- En MÉTANOT, une liste d'éléments i_1, i_2, ..., i_n est représentée par

  $<i_1, i_2, \ldots, i_n>$;

  la liste vide est représentée par `<>`.
- Le `k`-ème élément d'une liste `l` est noté `l(k)`.
- Les opérateurs suivants sont définis pour une liste d'objets `l`:
  - `l.EMPTY` détermine si une liste est vide;
  - `l.LENGTH` détermine le nombre d'éléments de la liste;
  - `l.FIRST` représente le premier élément de la liste;
  - `l.LAST` représente le dernier élément de la liste;
  - `l.HEAD` représente les (`l.LENGTH-1`) premiers éléments de la liste;
  - `l.TAIL` représente les (`l.LENGTH-1`) derniers éléments de la liste;
- La concaténation de deux listes `l1` et `l2` est notés `l1++l2`

EXPRESSIONS SYNTAXIQUES ABSTRAITES (SUITE)

Expressions relatives aux listes

Considérons une production de la catégorie liste, par exemple

`Compound` $\triangleq$ `Instruction`*

- pour décrire un spécimen à partir de la liste des composants, on écrit

  `instr` $\triangleq$ `Compound(<i1,i2,…,in>)`

  où `i1,i2,…,in` représentent des spécimens de type `Instruction`;
- pour décrire le `k`-ème composant d'un spécimen de type liste, on écrit

  `ik` $\triangleq$ `instr(k)`

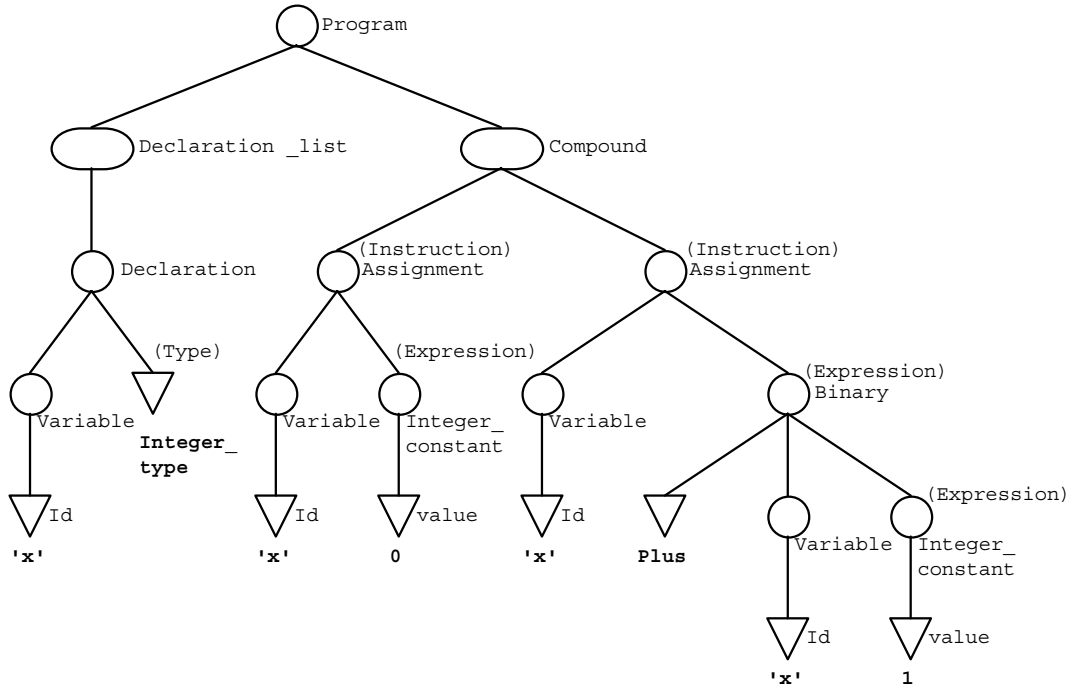
  où `instr` est spécimen de type liste et `k` un entier compris entre `1` et `instr.LENGTH`.

Exemple d'un spécimen de GRAAL

```
var1  ≙ Variable(id: 'x')
decl1 ≙ Declaration_list
             (<Declaration(v:var1; t:Type(Integer_type))>)
exp1  ≙ Expression(Constant(Integer_constant(value:0)))
exp2  ≙ Expression(var1)
exp3  ≙ Expression(Constant(Integer_constant(value:1)))
exp4  ≙ Expression(Binary(term1: exp2;
                          term2: exp3;
                          op: Operator(Arithmetic_op(plus))))
```

```
inst1 ≜ Instruction(Assignment(target:var1; source:exp1))
inst2 ≜ Instruction(Assignment(target:var1; source:exp4))
inst3 ≜ Instruction(Compound(<inst1,inst2>))
prog  ≜ Program(decpart:decl1; body:inst3)
```

ARBRE SYNTAXIQUE ABSTRAIT



SPÉCIFICATION DE LA SYNTAXE CONCRÈTE

Principe

- La syntaxe concrète d'un langage de programmation peut être décrite à partir de la syntaxe abstraite.
- La syntaxe concrète peut être formalisée au moyen de fonctions qui associent aux spécimen des chaînes de caractères.

Illustration sur les nombres entiers

Les nombres entiers sont définis par la grammaire

```
Number ≜ Digit+
Digit  ≜ Zero | One | Two | Three | Four | Five | Six |
         Seven | Eight | Nine
```

La syntaxe concrète peut être spécifiée au moyen de deux fonctions:

$concrete_{Number}$: Number -> **S**
$concrete_{Digit}$: Digit -> **S**

Illustration sur les nombres entiers (suite)

Les fonctions $\texttt{concrete}_{\texttt{Digit}}$ et $\texttt{concrete}_{\texttt{Number}}$ sont définies comme suit:

```
concrete_Digit(d:Digit) ≜
    case d of
      Zero => "0" | One => "1" | Two => "2" | Three => "3" |
      Four => "4" | Five => "5" | Six => "6" |
      Seven => "7" | Eight => "8" | Nine => "9"
    end

concrete_Number(n:Number) ≜
    if n.EMPTY then ""
    else concrete_Number(n.TAIL)//concrete_Digit(n.FIRST) end
```

où `//` représente l'opérateur de concaténation sur les chaînes de caractères.

FORMALISATION DE LA SÉMANTIQUE

Sémantique statique vs. sémantique dynamique

- La sémantique statique décrit les contraintes structurelles qui ne sont pas décrites par la syntaxe, comme par exemple:
  - un identificateur ne peut pas être déclaré deux fois;
  - une expression représentant le test d'une instruction conditionnelle ou d'une boucle doit être de type booléen;
  - etc.
- La sémantique dynamique décrit l'effet des programmes lors de leur exécution.

SPÉCIFICATION DE LA SÉMANTIQUE STATIQUE

Principe

- La sémantique statique d'un langage peut être décrite à partir de la syntaxe abstraite au moyen de fonctions de validités qui associent aux spécimens des valeurs booléennes.

Illustration sur les nombres entiers

- La contrainte qu'un nombre entier ne commence pas par des zéros peut être formalisée de la manière suivante:

```
validity_Number : Number -> B
validity_Number(n:Number) ≜
    case n.LAST of
      Zero => false |
      One, Two, ... , Nine => true
    end
```

SPÉCIFICATION DE LA SÉMANTIQUE DYNAMIQUE

Principe

- La sémantique dynamique d'un langage peut être décrite à partir de la syntaxe abstraite au moyen de fonctions sémantiques:
  - dont les domaines de départ sont dits *domaines syntaxiques* et constitués par les spécimens;
  - dont les domaines d'arrivée sont dits *domaines sémantiques* et représentent des propriétés ou des valeurs.

Illustration sur les nombres entiers

- La valeur d'un nombre entier peut être formalisée au moyen des fonctions suivantes:

```
value_Digit: Digit -> N
value_Digit(d:Digit) ≜
     case d of
        Zero => 0 | One => 1 | Two => 2 | Three => 3 |
        Four => 4 | Five => 5 | Six => 6 | Seven => 7 |
        Eight => 8 | Nine => 9
     end
value_Number(n:Number) : Number -> N
value_Number(n:Number) ≜
     if n.EMPTY then 0
     else value_Digit(n.FIRST) + 10*value_Number(n.TAIL) end
```

SYNTAXE CONCRÈTE ET SÉMANTIQUE

Exercice 1

En partant de sa grammaire abstraite, décrire la syntaxe concrète de Métanot .

Exercice 2

Graal ne contient pas l'opérateur unaire `Not` (négation logique). Définir une fonction

```
Not : Expression -> Expression
```

telle que pour toute expression logique `e`, `Not(e)` est une expression dont la sémantique correspond à la négation de `e`.